

Introduction to Scapegoat plugin

What's Scapegoat

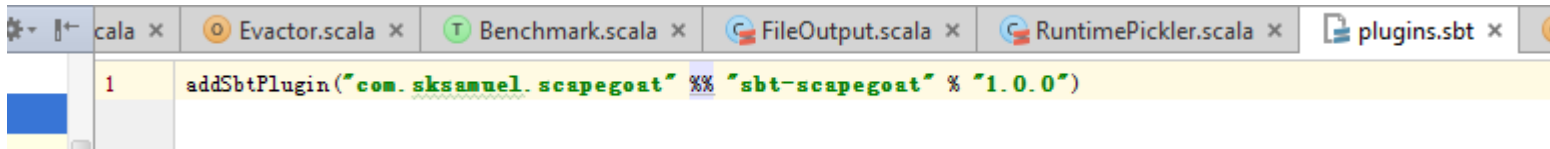
- Scapegoat is a Scala static code analyzer (similar to Linter)
- A static code analyzer is a tool that
 - flag suspicious language usage in code.
- This can include behavior likely to
 - lead or bugs,
 - non idiomatic usage of a language
 - code that doesn't conform to specified style guidelines.

Prerequisites

- Scapegoat is a Scala compiler plugin, so you can run it with SBT.
- It is an auto plugin, so you need SBT 0.13.5 or higher.
- Scapegoat only works with Scala 2.11.x, so **don't try it on Scala 2.10.x projects.**

How to use Scapegoat plugin

- 1. Add the following line into *project/plugins.sbt* or other .sbt files
 - `addSbtPlugin("com.sksamuel.scapegoat" %% "sbt-scapegoat" % "1.0.0")`



The screenshot shows an IDE window with several tabs: 'cala', 'Evactor.scala', 'Benchmark.scala', 'FileOutput.scala', 'RuntimePickler.scala', and 'plugins.sbt'. The 'plugins.sbt' tab is active, displaying the following code on line 1:

```
addSbtPlugin("com.sksamuel.scapegoat" %% "sbt-scapegoat" % "1.0.0")
```

How to use Scapegoat plugin

- 2. Build your project with *sbt*, and use *scapegoat* task to generate report

```
> sbt
```

```
Java HotSpot(TM) 64-Bit Server VM warning...
```

```
E:\Scala\Codes\scala_pickling_0.10.x\pickling-0.10.x>sbt
```

```
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed in 8.0
```

```
[info] Loading project definition from E:\Scala\Codes\scala_pickling_0.10.x\pickling-0.10.x\project
```

```
[info] Compiling 3 Scala sources to E:\Scala\Codes\scala_pickling_0.10.x\pickling-0.10.x\project\target\scala-2.10\sbt-0.13\classes...
```

```
[info] Set current project to Scala Pickling (in build file:/E:/Scala/Codes/scala_pickling_0.10.x/pickling-0.10.x/)
```

How to use Scapegoat plugin

- 3. Run *Scapegoat* task

> scapegoat

[info] Resolving...

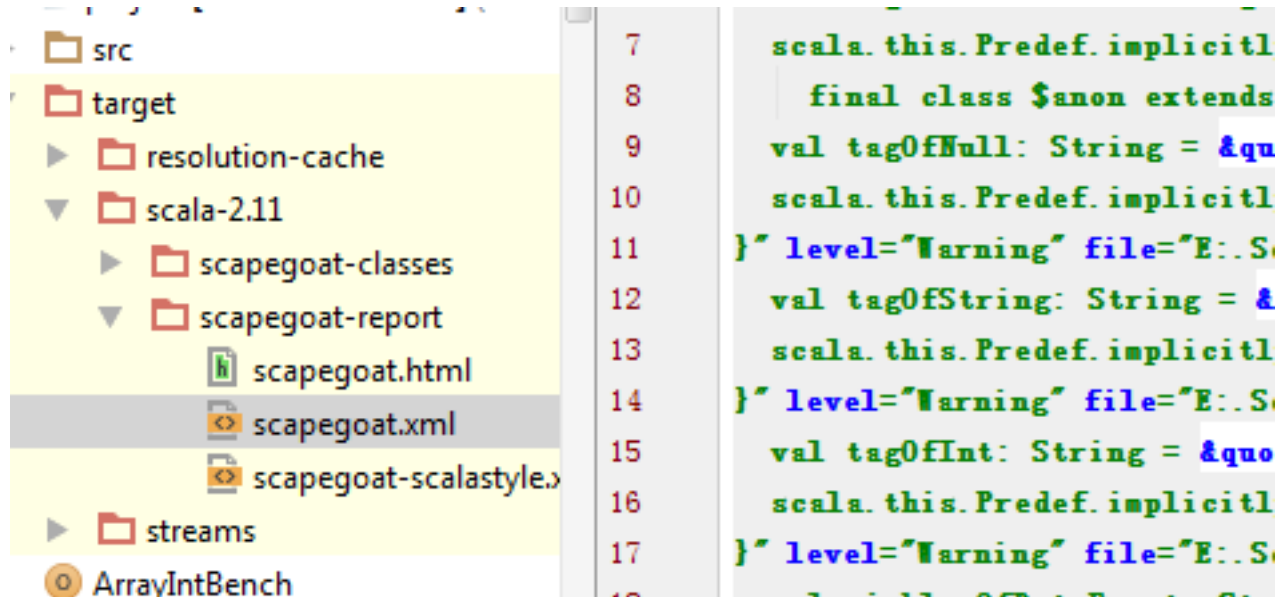
[info] [scapegoat] setting output dir to ...

[info] [scapegoat]: 55 activated inspections

```
> scapegoat
[info] Updating {file:/E:/Scala/Codes/scala_pickling_0.10.x/pickling-0.10.x/}core...
[info] Resolving org.scala-lang#scala-compiler;2.11.6 ...
[info] [scapegoat] setting output dir to [E:\Scala\Codes\scala_pickling_0.10.x\pickling-0.10.x\benchmark\target\scala-2.11\scapegoat-report]
[info] [scapegoat] setting output dir to [E:\Scala\Codes\scala_pickling_0.10.x\pickling-0.10.x\target\scala-2.11\scapegoat-report]
[info] [scapegoat] setting output dir to [E:\Scala\Codes\scala_pickling_0.10.x\pickling-0.10.x\sandbox\target\scala-2.11\scapegoat-report]
[info] Compiling 53 Scala sources and 1 Java source to E:\Scala\Codes\scala_pickling_0.10.x\pickling-0.10.x\core\target\scala-2.11\scapegoat-classes...
[info] Compiling 53 Scala sources and 1 Java source to E:\Scala\Codes\scala_pickling_0.10.x\pickling-0.10.x\core\target\scala-2.11\classes...
[info] [info] [scapegoat] 12 activated inspections
[info] [info] [scapegoat] 12 activated inspections
```

How to use Scapegoat plugin

4. Get the reports inside *target/scala-2.11/scapegoat-report*.



Analyzing Scala code

- Scala Runtime Reflection
- Given a type or instance of some object at runtime, reflection is the ability to:
 - inspect the type of that object, including generic types,
 - to instantiate new objects,
 - or to access or invoke members of that object.

```
1. scala> import scala.reflect.runtime.{universe => ru}
2. import scala.reflect.runtime.{universe=>ru}
3.
4. scala> val l = List(1,2,3)
5. l: List[Int] = List(1, 2, 3)
6.
7. scala> def getTypeTag[T: ru.TypeTag](obj: T) = ru.typeTag[T]
8.   getTypeTag: [T](obj: T)(implicit evidence$1: ru.TypeTag[T])ru.TypeTag[T]
9.
10. scala> val theType = getTypeTag(l).tpe
11. theType: ru.Type = List[Int]
```


Scala Reflection

- Trees are the basis of Scala's abstract syntax which is used to represent programs. They are also called abstract syntax trees and commonly abbreviated as ASTs.
- When working with runtime reflection, one need not construct trees manually.
- Via method reify (should be preferred wherever possible), one can create AST of the code.

How to use reify

```
1. scala> import scala.reflect.runtime.universe._
2. import scala.reflect.runtime.universe._
3.
4. scala> { val tree = reify(println(2)).tree; showRaw(tree) }
5. res0: String = Apply(Select(Select(This(TypeName("scala")),
    TermName("Predef")), TermName("println")), List(Literal(Constant(2))))
```

```
1. scala> val x = reify(2)
2. x: scala.reflect.runtime.universe.Expr[Int(2)] = Expr[Int(2)](2)
3.
4. scala> reify(println(x.splice))
5. res1: scala.reflect.runtime.universe.Expr[Unit] = Expr[Unit]
    (scala.this.Predef.println(2))
```

How to use **reify**

- **import** scala.reflect.runtime.*universe*._
val prog = reify {
 - *//Your code here*
- }
- **val** rawStr = showRaw(prog)
- Eg:
- rawStr = Apply(Select(Ident(TermName("x")), TermName("\$plus")), List(Literal(Constant(2))))

Traverse AST via Traverser

- To use a Traverser:
 - simply subclass Traverser
 - override method traverse.
- In doing so, you can simply provide custom logic to handle only the cases you're interested in.

```
scala> object traverser extends Traverser {  
  |   var applies = List[Apply]()  
  |   override def traverse(tree: Tree): Unit = tree match {  
  |     case app @ Apply(fun, args) =>  
  |       applies = app :: applies  
  |       super.traverse(fun)  
  |       super.traverseTrees(args)  
  |     case _ => super.traverse(tree)  
  |   }  
  | }  
defined module traverser
```

Scapegoat plugin

- Use Scala Runtime Reflection to generate AST
- Use customized traverser to retrieve branches you are interested

```
class TemplateMatcher extends Inspection {  
  def inspector(context: InspectionContext): Inspector = new Inspector(context) {  
  
    import context.global._  
  
    override def postTypeTraverser = Some apply new context.Traverser {  
  
      override def inspect(tree: Tree): Unit = {  
        tree match {  
          case _ =>  
            continue(tree)  
            context.warn("Some warning", tree.pos, Levels.Warning,  
              "some description" + tree.toString().take(500), TemplateMatcher.this)  
        }  
      }  
    }  
  }  
}
```

How to create detectors(traversers)

- There is a sample class named “TemplateMatcher” provided in package *com.sksamuel.scapegoat.inspections.genAST*
- You can implement your own detector here, or treat it as the template of your own traversers.
- There are many other detectors implemented. You can learn from them as well.

Bug Patterns

- To implement a detector, you need a pattern first.
- Suppose there is a pattern named “MissingStringInterpolation”
- In Scala, you can insert string value of a variable directly into a string with an “s” ahead of the string.
- But people sometimes forget about it.

```
object Test {
```

```
  //Missing “s” here, should be s”this...” !
```

```
  val str = "this is my $interpolated string lookalike"
```

```
}
```

Bug Patterns

- Hence, you've got a bug pattern, and you should generate the AST of some samples of the pattern.
- Use *GenAST* in package `genAST` to get the AST of this sample

```
import scala.reflect.runtime.universe._  
val prog = reify { //Your code here  
  object Test {  
    | val str = "this is my $interpolated string lookalike"  
  }  
}  
  
val rawStr = showRaw(prog)
```


Bug Patterns

```
ModuleDef(  
  Modifiers(),  
  TermName("Test"),  
  Template(  
    List(Ident(TypeName("AnyRef"))),  
    noSelfType,  
    List(  
      DefDef(  
        Modifiers(),  
        termNames.CONSTRUCTOR,  
        List(),  
        List(List()),  
        TypeTree(),  
        Block(  
          List(  
            Apply(  
              Select(Super(This(typeNames.EMPTY), typeNames.EMPTY), termNames.CONSTRUCTOR),  
              List()  
            ),  
            Literal(Constant(())))  
          ),  
          Literal(Constant(())))  
        ),  
      ValDef(  
        Modifiers(),  
        TermName("str"),  
        TypeTree(),  
        Literal(Constant("this is my $interpolated string lookalike"))))  
    )  
  )  
)
```

Bug Patterns

- Focus on the part you are interested.

```
ValDef(  
  Modifiers(),  
  TermName("str"),  
  TypeTree(),  
  Literal(Constant("this is my $interpolated string lookalike"))  
)
```

- which is for

```
val str = "this is my $interpolated string lookalike"
```

- So, our Pattern AST would be like
 - *Literal(Constant(str)) if str contains "\$"*

Bug Patterns

- And the detector would be like

```
override def postTyperTraverser = Some apply new context.Traverser {  
  
  override def inspect(tree: Tree): Unit = {  
    //Your implementations here.  
    tree match {  
      case Literal(Constant(str: String)) if str contains "$" =>  
        //This is an example of how to raise warnings.  
        context.warn("Missing String interpolation", tree.pos, Levels.Warning,  
          "Look like you missed a string interpolation at " + tree.toString().take(500), TemplateMatcher.this)  
      case _ =>  
        continue(tree)  
    }  
  }  
}
```

Bug Pattern

- You may generalize bug pattern from
 - The mistakes you've made
 - Searching for bug patterns on Google or Baidu
 - Some issue trackers of famous Scala frameworks or projects
 - Questions on StackOverflow
 - The book "Scala for the Impatient"
 - Projects in Scala and SBT Github repository
 - Documentation of Scala-lang (<http://www.scala-lang.org/>)
 - Open source projects on Github (Recommended)

Bug Pattern

- But this is far from satisfying.
- Here is another example:
 - ```
object Test {
 val str = "$.ajax{...}."
}
```
- And now, we encounter a **false positive**.

# False Positive

- Almost every detector has some kind of false positives.
- We should eliminate them in our final product.
- Use ScalaTest to test your detectors.

# ScalaTest

- Scapegoat has already complete the testing framework for us.
- We are only required to fill in our own test cases.
- There is also a sample test class for TemplateMatcher in package *com.sksamuel.scapegoat.inspections.myTraversers*

```
"LooksLikeInterpolatedString" = {
 "should report warning" = {
 "for string containing $var" in {
 val code = """object Test {
 val mystr = "some string"
 val str = $mystr
 } """
 compileCodeSnippet(code)
 compiler.scapegoat.feedback.warnings.size shouldBe 1
 }
 }
}
```

# ScalaTest

- To eliminate false positives, (currently) the best way is to test your traversers on open source projects.
- There is a list of suggested projects in the Caution section that you might want to try.



# Enable your own detectors

- Add new instance of your detector in
  - *com.sksamuel.scapegoat.ScapegoatConfig*
- Comment the default detectors.
- No more work needed.

# Apply

- If you don't want to publish your project, follow these instructions to add your own version of *scapegoat* into ivy cache.
- 1. Compile the Scapegoat project, and pack them into a .jar file
- 2. Copy the jar to
  - *(Path to .ivy2)\.ivy2\cache\com.sksamuel.scapegoat\scalac-scapegoat-plugin\_2.11\jars*
- 3. Replace the jar file with your own one. Make sure you didn't change the file name.
  - *scalac-scapegoat-plugin\_2.11-1.0.0.jar*
- And now you can use scapegoat task with only your own detectors enabled.

# Caution

- Scapegoat is NOT a perfect framework.
  - You might encounter “Missing dependencies” on *scapegoat* task, but SBT compiles succeeds anyway.
  - Here is a list of projects on which *scapegoat* runs normally.

|                                       |                                        |
|---------------------------------------|----------------------------------------|
| adamw_macwire                         | finagle_finch                          |
| daniel-trinh_scalariform              | maxcellent_lamma                       |
| p2t2_figaro                           | mesosphere_chaos                       |
| dickwall_subcut                       | ngocdaoanh_netcaty_master              |
| dispatch_reboot                       | nafg_reactive_v0.4.0                   |
| dylemma_scala-frp                     | novus_salat__master                    |
| fehmicansaglam_tepkin                 | ReactiveMongo_ReactiveMongo_master     |
| ReactiveX_RxScala_0.x                 | sksamuel_elastic4s_master              |
| romainreuillon_mgo_master             | squeryl_squeryl_master                 |
| SandroGrzicic_ScalaBuff_master        | xitrum-framework_scala-xgettext_master |
| scala_async_master                    | twitter_finatra_master                 |
| scala_pickling_0.10.x                 | unfiltered_unfiltered_0.9.0            |
| scala_scala-parser-combinators_master | lift-framework_master                  |
| sirthias_scala-ssh_master             |                                        |

# Caution

- Although you can't test scapegoat out of these projects, you can still generalize pattern from other projects.
- Downloading dependencies takes lots of time. Please be patient.
- A simple way to find bugs is to read commit messages from projects in Github. Commits with message containing keywords like "Fix" or "Bug" are likely to be bugs. But lots of them cannot be generalized as a pattern. You need to filter them out.

# References

- Scapegoat plugin
  - <https://github.com/sksamuel/scalac-scapegoat-plugin>
- Scala Reflection
  - <http://docs.scala-lang.org/overviews/reflection/symbols-trees-types.html>
- Collection of Scala projects on Github
  - <https://github.com/lauris/awesome-scala>
- Spark Issue Tracker
  - <https://issues.apache.org/jira/browse/spark/?selectedTab=com.atlassian.jira.jira-projects-plugin:issues-panel>
- Scala Issues
  - <https://issues.scala-lang.org/secure/Dashboard.jspa>

# References

- Similar projects
  - <http://stackoverflow.com/questions/1598882/are-there-any-tools-for-performing-static-analysis-of-scala-code>
  - <https://github.com/sksamuel/scalac-scapegoat-plugin> (#other static analysis tools)
- Scala on Coursera
  - <https://class.coursera.org/progfun-005>

Thanks